Arrays and I-structures

Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.

October 12, 2006

Arrays

Cache for function values on a regular subdomain

\[ x = \text{mkArray} \ (1, \ n) \ f \]

means \( x!i = (f \ i) \quad 1 \leq i \leq n \)

Selection: \( x!i \) returns the value of the \( i^{th} \) slot

Bounds: \( \text{(bounds} \ x) \) returns the tuple containing the bounds
Efficiency is the Motivation for Arrays

(f \ i) is computed once and stored

x!i is simply a fetch of a precomputed value and should take constant time

Index Type Class

Arrays can be indexed by any type that can be regarded as having a contiguous enumerable range

```haskell
class Ix a where
    range :: (a,a) -> [a]
    index :: (a,a) -> a -> Int
    inRange :: (a,a) -> a -> Bool
```

range: Returns the list of index elements between a lower and an upper bound

index: Given a range and an index, it returns an integer specifying the position of the index in the range based on 0

inRange: Tests if an index is in the range
Examples of Index Type

```haskell
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat

An index function may be defined as follows:

```haskell
index (Sun,Sat) Wed = 3
index (Sun,Sat) Sat = 6
...
```

A two dimensional space may be indexed as follows:

```haskell
index ((li, lj), (ui, uj)) (i, j) =
    (i-li)*((uj-lj)+1) + j - lj
```

This indexing function enumerates the space in the row major order.

Array: An Abstract Datatype

```haskell
module Array (Array, mkArray, (!), bounds)
where

    infix 9 (!)

data (Ix a) => Array a t
mkArray :: (Ix a) => (a,a) -> (a -> t) -> (Array a t)
(!) :: (Ix a) => (Array a t) -> a -> t
bounds :: (Ix a) => (Array a t) -> (a,a)

Thus,

type ArrayI t = Array Int t

type MatrixI t = Array (Int,Int) t
```
Higher Dimensional Arrays

\[ x = \text{mkArray } ((l_1, l_2), (u_1, u_2)) f \]

means \[ x!((i,j)) = f(i,j) \quad 11 \leq i \leq u_1 \]
\[ 12 \leq j \leq u_2 \]

Type

\[ x :: (\text{Array} \ (\text{Int}, \text{Int}) \ t) \]

Assuming

\[ f :: (\text{Int}, \text{Int}) \rightarrow t \]

\text{mkArray} will work for higher dimensional matrices as well.

Array of Arrays

\((\text{Array} \ \text{Int} \ (\text{Array} \ \text{Int} \ t)) \neq (\text{Array} \ (\text{Int}, \text{Int}) \ t)\)

This allows flexibility in the implementation of higher dimensional arrays.
Matrices

\[
\text{add } (i,j) = i + j
\]

\[
\text{mkArray } ((1,1),(n,n)) \text{ add }
\]

\[
\text{mkArray } ((1,1),(n,n)) \text{ add }
\]

\[
\text{transpose a = let}
\]

\[
((l1,l2),(u1,u2)) = \text{bounds a}
\]

\[
f (i,j) = a!(j,i)
\]

\[
in
\]

\[
\text{mkArray } ((l2,l1),(u2,u1)) f
\]

\[
\text{mkArray } ((l2,l1),(u2,u1)) f
\]
The Wavefront Example

\[ x_{i,j} = x_{i-1,j} + x_{i,j-1} \]

\[ x = \text{mkArray } ((1,1),(n,n)) (f \ x) \]

\[ f \ x \ (i, j) = \begin{cases} 
1 & \text{if } i == 1 \\
1 & \text{if } j == 1 \\
\text{else } x!(i-1,j) + x!(i,j-1) & \text{else}
\end{cases} \]

Compute the least fix point.

\[ x = \text{mkArray } ((1,1),(n,n)) (f \ x) \]

\[ f \ x \ (i, j) = \begin{cases} 
1 & \text{if } i == 1 \\
1 & \text{if } j == 1 \\
\text{else } x!(i-1,j) + x!(i,j-1) & \text{else}
\end{cases} \]
Array Comprehension

A special function to turn a list of (index,value) pairs into an array

```
array :: (Ix a) => (a,a) -> [(a,t)] -> (Array a t)
array ebound
  [(ie1,e1) | gen-pred, ..]
  ++ [(ie2,e2) | gen-pred, ..] ++ ..
```

Thus,

```
mkArray (l,u) f =
  array (l,u) [(j,(f j)) | j <- range(l,u)]
```

List comprehensions and function array provide flexibility in constructing arrays, and the compiler can implement them efficiently.

duplicates?

---

Array Comprehension: Wavefront

```
\[ x[i,j] = x[i-1,j] + x[i,j-1] \]
```

```
x = array ((1,1),(n,n))
  [(i-1,1), (1)]
  ++ [(i,1), 1 | i <- [2..n]]
  ++ [(1,j), 1 | j <- [2..n]]
  ++ [(i,j), x!(i-1,j) + x!(i,j-1))
  | i <- [2..n],
  | j <- [2..n]
```

---
Duplicates

- Haskell Semantics:
  - Enumerate the whole index range and return bottom if any duplicate indices are found

- pH Semantics:
  - Only the duplicated elements are bottom, no the whole array

- Haskell semantics are motivated by lazy evaluation and awful for parallel implementation; pH semantics are preferable if all the elements of the array are going to be computed

Another Issue: Computed Indices

<table>
<thead>
<tr>
<th>2</th>
<th>5</th>
<th>6</th>
<th>1</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Inverse permutation

\[ y \left( x ! i \right) = i \]

\[
\text{find } x \ i = \\
\text{let } \% \text{ find } j \text{ such that } x!j = i \\
\text{step } j = \text{if } x!j == i \text{ then } j \\
\text{else step } j+1 \\
\text{in step 1}
\]

\[ y = \text{mkArray} \ (1,n) \ (\text{find } x) \]

How many comparisons? Can we do better?

\[ y = \text{array} \ (1,n) \ [[[ x!i \ , \ i ]] \mid i <- [1..n]] \]
I-structures

In functional data structures, a single construct specifies:
- The shape of the data structure
- The value of its components

These two aspects are specified separately using I-structures
\[ \Rightarrow \text{efficiency} \]
\[ \Rightarrow \text{parallelism} \]

I-structures preserve determinacy but are not functional!

I-Arrays

- Allocation expression
  \[ \text{iArray} \ (1,n) \ [\] \]

  \[
  \begin{array}{cccc}
  1 & 2 & \ldots & n \\
  \bot & \bot & \cdots & \bot \\
  \end{array}
  \]

- Assignment
  \[ \text{iAStore} \ a \ 2 \ 5 \]
  \[ \text{or} \ a!2 := 5 \]

  provided the previous content was \( \bot \)
  "The single assignment restriction."

- Selection expression
  \[ a!2 \rightarrow 5 \]
  (\( \bot \) means empty)
Computed Indices Using I-structures

Inverse permutation
\( y \cdot (x \cdot i) = i \)

让 
\[
\text{let } y = \text{iArray}(1,n) [] \\
\_ = \text{for } i <- [1..n] \text{ do} \\
\_ = \text{iAStore } y \ (x!i) \ i \\
\text{finally } () \ % \text{ unit data type} \\
\text{in} \\
y
\]

What if \( x \) contains a duplicate?

Multiple-Store Error

Multiple assignments to an iArray slot cause a multiple store error

A program with exposed store error is supposed to blow up!

Program --> T

The Top represents a contradiction
The Unit Type

\[ \text{data } () = () \]

means we cannot do much with an object of the unit type. However, it does allow us to drop \'_ ='_

\[
\text{let y = iArray (1,n) []}
\]
\[
\text{for i <- [1..n] do}
\]
\[
\text{iAStore y (x!i) i}
\]
\[
\text{finally () -- unit data type}
\]
\[
\text{in}
\]
\[
y
\]

For better syntax replace

\[ \text{iAStore y (x!i) i by y!(x!i) := i} \]

I-Cell

\[ \text{data ICell a = ICell \{contents :: \_ : a\}} \]

\text{Constructor}

\[ \text{ICell :: a -> ICell a} \]
\[ \text{ICell e}
\]
\[ \text{or}
\]
\[ \text{ICell \{contents = e\}} \]

\text{or create an empty cell and fill it}

\[ \text{ic = ICell \{\}}
\]
\[ \text{contents ic := e}
\]

\text{Selector}

\[ \text{contents ic}
\]
\[ \text{or}
\]
\[ \text{case ic of}
\]
\[ \text{ICell x -> ... x ...} \]
An Array of ICells

Example: Rearrange an array such that the negative numbers precede the positive numbers

\[
\begin{array}{cccccc}
2 & 8 & -3 & 14 & 2 & 7 \\
-3 & -5 & 2 & 8 & 14 & 2 & 7 \\
\end{array}
\]

Functional solutions are not efficient

```haskell
let y = array (1,n) [(i,ICell {})| i<-[1..n]]
(l,r) = (0,n+1)
final_r = for j <- [1..n] do
  (l’,r’,k) =
    if (x!j >= 0)
      then (l,r-1,r-1)
      else (l+1,r,l+1)
    contents (y!k) := x!j
  next l = l’
  next r = r’
finally r
in (y, final_r)
```

Type Issues

In the previous example

```
x :: Array Int
y :: Array (ICell Int)
```

1. IArray data type eliminates this extra level of indirection

2. The type of a functional array (Array) is different from the type of an IArray.

However, an IArray behaves like a functional Array after all its elements have been filled

We provide a primitive function for this conversion

```
cvt_IArray_to_Array ia -> a
```
Types Issue (cont.)

Hindley-Milner type system has to be extended to deal with I-structures

⇒ ref type -- requires new rules

\[ don't \ have \ time \ to \ get \ into \ this \ \ldots \]
Array Comprehensions:
*a packaging of I-structures*

array dimension

\[((i1,e1) | x \leftarrow xs, y \leftarrow ys)\] ++ \[((i2,e2) | z \leftarrow zs)\]

translated into

\[
\begin{array}{l}
\text{let } a = \text{iArray dimension } [] \\
\quad \text{for } x \leftarrow xs \\
\quad \quad \text{for } y \leftarrow ys \\
\quad \quad \quad a!i1 := e1 \\
\quad \quad \quad \text{finally } () \\
\quad \quad \text{finally } () \\
\quad \quad \text{for } z \leftarrow zs \\
\quad \quad \quad a!i2 := e2 \\
\quad \quad \quad \text{finally } () \\
\quad \text{in cvt_IArray_to_Array a}
\end{array}
\]

We have used pH syntax but it is trivial to translate this into Haskell syntax.

I-structures are *non functional*

\[
f \ x \ y = \text{let } x!1 := 10 \\
\quad y!1 := 20 \\
\quad \text{in } ()
\]

\[
\begin{array}{l}
\text{let } x = \text{iArray } (1,2) \ [] \\
\text{in } f \ x \ x
\end{array}
\]

= \[
\begin{array}{l}
f \ (\text{iArray } (1,2) \ []) \ (\text{iArray } (1,2) \ [])
\end{array}
\]
The example

\[
\begin{align*}
f \ x \ y &= \text{let} \ x!1 := 10 \\
& \quad \ y!1 := 20 \\
& \quad \text{in} \ ()
\end{align*}
\]

let
\[
x = \text{iArray} (1,2) []
\]
in
\[
f \ x \ x
\]
⇓
let
\[
x = \text{iArray} (1,2) []
\]
\[
x!1 := 10 \\
x!1 := 20
\]
⇓
"blow up"

\[
\begin{align*}
f \ (\text{iArray} (1,2) []) \\
& (\text{iArray} (1,2) [])
\end{align*}
\]
⇓
let
\[
t1 = \text{iArray} (1,2) []
\]
\[
t2 = \text{iArray} (1,2) []
\]
\[
t1!1 := 10 \\
t2!1 := 20 \\
in \ ()
\]

We have finally slipped into
- parallelism issues
- side-effects

More on these issues after the quiz